LA-UR-20-24900
Approved for public release; distribution is unlimited.

Title: Accelerate M-TIP on GPUs and deploy to Summit and NERSC-9 (against simulated data) WBS 2.2.4.05 ExaFEL, Milestone ADSE13-199

Author(s): Kommera, Pranay Reddy
Ramakrishnaiah, Vinay Bharadwaj
Sweeney, Christine Marie

Intended for: Report

Issued: 2021-11-16 (rev.1)

ECP-U-2017-XXX

Accelerate M-TIP on GPUs and deploy to Summit and NERSC-9 (against simulated data)
WBS 2.2.4.05 ExaFEL, Milestone ADSE13-199

Authors:

Pranay Kommera, Vinay Ramakrishnaiah, Christine Sweeney

Author Affiliation: LANL

7/2/20

ECP-U-2017-XXX

ECP Milestone Report
Accelerate M-TIP on GPUs and deploy to Summit and NERSC-9 (against simulated data)
WBS 2.2.4.05, Milestone ADSE13-199

Office of Advanced Scientific Computing Research
Office of Science
US Department of Energy

Office of Advanced Simulation and Computing
National Nuclear Security Administration
US Department of Energy

July 2, 2020

ECP Milestone Report
Accelerate M-TIP on GPUs and deploy to Summit and NERSC-9 (against simulated data)

WBS 2.2.4.05, Milestone ADSE13-199

APPROVALS

Submitted by:

Christine Sweeney, Scientist Los Alamos National Laboratory	Date

Concurrence:

Author, Title Affiliation	Date

Approval:

Author, Title Affiliation	Date

REVISION LOG

Version	Creation Date	Description	Approval Date
1.0	7/2/20	Original	

CONTENTS

APPROVALS iii

REVISION LOG v

CONTENTS vii

LIST OF FIGURES x

LIST OF TABLES x

EXECUTIVE SUMMARY xi

1. INTRODUCTION 1

2. MILESTONE OVERVIEW 3

2.1 DESCRIPTION 3

2.2 EXECUTION PLAN 3

2.3 COMPLETION CRITERIA 3

2.4 MILESTONE DEPENDENCIES 3

2.4.1 Milestone Predecessors 3

2.4.2 Milestone Successors 3

3. TECHNICAL WORK SCOPE, APPROACH, RESULTS 4

3.1 Work Scope 4

3.2 Datasets 4

3.3 Euclidean Distance and K-Nearest Neighbors 5

3.4 Sequential and multi-threaded Orientation Matching Algorithms and Implementations 5

3.5 CUDA-Based GPU AccelerateD Explorations of Orientation Matching Algorithms and Implementations 7

3.5.1 CrabCUDA 7

3.5.2 cuML 7

3.5.3 FAISS CUDA Version 7

3.5.4 Background to LANL CUDA Versions 7

3.5.5 LANL CUDA v3 and v5 Implementations 9

3.6 Summary of Orientation Matching Algorithms Explored and Implemented 11

3.7 I/O Optimization 12

3.8 Results 12

3.8.1 Orientation Matching 12

3.8.2 HDF5 15

4. RESOURCE REQUIREMENTS 17

5. CONCLUSIONS AND FUTURE WORK 17

6. ACKNOWLEDGMENTS 17

7. REFERENCES 18

LIST OF FIGURES

Figure 1 Single particle imaging experiment. Image courtesy J. Donatelli. ... 1
Figure 2 Steps in the M-TIP algorithm. Image courtesy J. Donatelli ... 1
Figure 3 Diagram of M-TIP algorithm. Image courtesy J. Donatelli. ... 2
Figure 4 6NYF. Image courtesy [4]. ... 4
Figure 5 2CEX. Image courtesy [5]. ... 4
Figure 6 Illustration of the CUDA tiling method. ... 8
Figure 7 Data communication between GPUs for k-Nearest Neighbots algorithm ... 9
Figure 8 Data distribution scheme for reference images. ... 10
Figure 9 Comparison of various implementations of orientation matching algorithm on different image sizes. ... 12
Figure 10 Performance of LANL CUDA unweighted algorithm on Summit and Cori ... 13
Figure 11 Performance comparison of LANL CUDA weighted algorithm on Summit and Cori ... 14
Figure 12 Strong scaling of LANL CUDA implementation on Summit. ... 14
Figure 13 Strong Scaling of LANL CUDA version over MPI Ranks on a single node. ... 15
Figure 14 HDF5 read times with increasing MPI processes. ... 15
Figure 17 HDF5 read times on Cori (np=64) ... 17

LIST OF TABLES

Table 1 Summary of algorithms ... 11

EXECUTIVE SUMMARY

We evaluated the orientation matching step in the M-TIP SPI workflow for potential offloading to accelerators. We ported the code to GPUs, benchmarked it, optimized and down-selected the best versions. The accelerated version of the orientation matching code that was developed at LANL (LANL GPU v3) is 34-55X faster than sequential, 2.4-4.9X faster than the fastest OpenMP open source version we found (FAISS OpenMP) and 1.5-4X faster than the fastest GPU open source version we found (FAISS GPU). Summit single-node GPU versions were somewhat faster than Cori GPU. Image size plays a role; mid-range image sizes take more time. The LANL CUDA multi-node, multi-GPU implementation shows mostly linear strong scaling. I/O also plays a large role; splitting data into parts improves read time and burst buffers dramatically improve read times. This work will be integrated into the M-TIP workflow as part of the next milestone ADSE13-193.

1. INTRODUCTION

The Multi-Tiered Iterative Phasing (M-TIP)[1] algorithm for single particle imaging (SPI) is used to reconstruct a 3D structure of a molecule from X-ray diffraction patterns of single particles of the molecule obtained in light source experiments at facilities such as LCLS using an X-ray free electron laser (XFEL). It is a novel algorithm used by the ExaFEL project, because it can reconstruct structural information from single-particle diffraction data by simultaneously determining the states, orientations, intensities, phases, and underlying structure in a single iterative procedure.

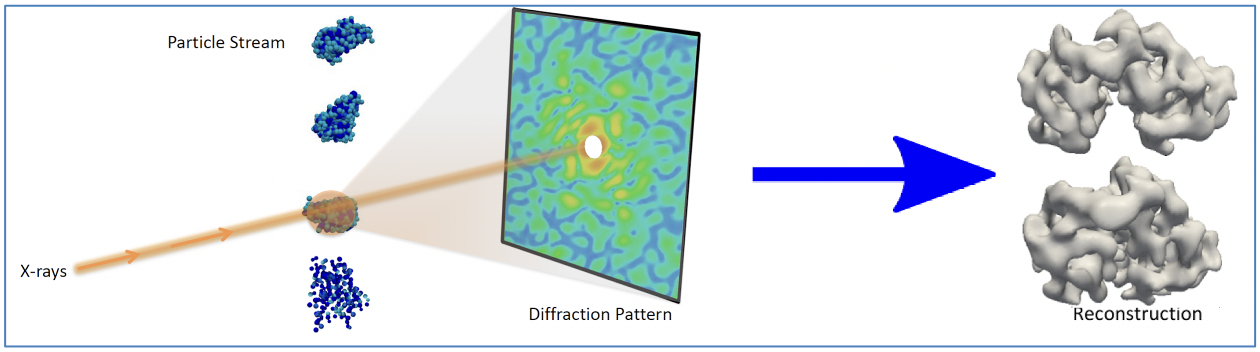


Figure 1 Single particle imaging experiment. Image courtesy J. Donatelli.

Particles of a molecule are streamed through the path of a coherent X-ray beam and diffraction patterns are captured on detectors behind the particle stream (Figure 1). Each X-ray diffraction pattern samples a slice of the molecule's 3D diffraction volume at a random orientation. Slices are combined using the M-TIP iterative algorithm to reconstruct the 3D structure. Steps in the algorithm include classification of the particle as to conformational state, determination of the orientation of the particle in space, assembly of all the diffraction patterns (slices) into a 3D diffraction volume and a determination of the phase, which yields the molecular structure (Figure 2).

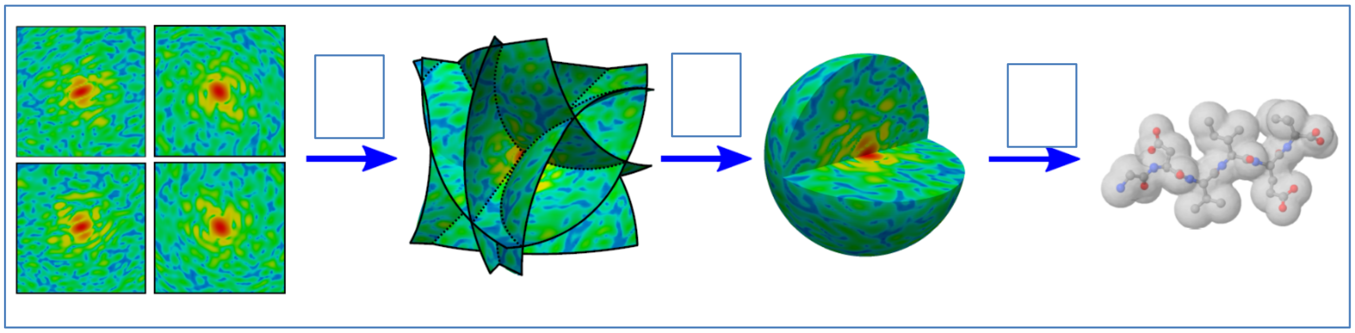
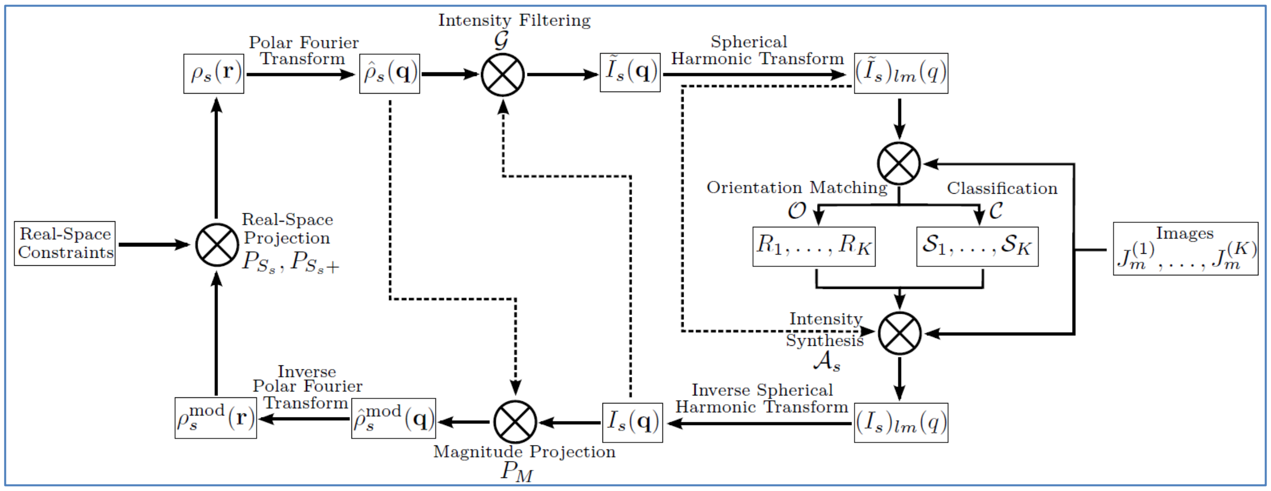


Figure 2 Steps in the M-TIP algorithm. Image courtesy J. Donatelli

The M-TIP algorithm for SPI is illustrated below in Figure 3 with the orientation matching step labelled. This figure shows that this step occurs on each iteration of the algorithm, thus its speed is critical to the performance of the M-TIP algorithm as a whole.



Intensity Filtering $\mathcal{G}$
Polar Fourier Transform
Spherical Harmonic Transform
$\rho_s(\mathbf{r})$
$\hat{\rho}_s(\mathbf{q})$
$\tilde{I}_s(\mathbf{q})$
$(\tilde{I}_s)_{lm}(q)$
Orientation Matching $\mathcal{O}$
Classification $\mathcal{C}$
$R_1, \dots, R_K$
$\mathcal{S}_1, \dots, \mathcal{S}_K$
Images $J_m^{(1)}, \dots, J_m^{(K)}$
Real-Space Constraints
Real-Space Projection P_{S_s}, P_{S_s+}
Intensity Synthesis $\mathcal{A}_s$
Inverse Polar Fourier Transform
$\rho_s^{\text{mod}}(\mathbf{r})$
$\hat{\rho}_s^{\text{mod}}(\mathbf{q})$
$I_s(\mathbf{q})$
Inverse Spherical Harmonic Transform
$(I_s)_{lm}(q)$
Magnitude Projection P_M

Figure 3 Diagram of M-TIP algorithm. Image courtesy J. Donatelli.

This milestone focuses on accelerating the orientation matching step in the M-TIP algorithm on GPU for high-performance execution on upcoming exascale computers.

Orientation matching involves comparing the similarity of images from the detector and finding the closest matching model. This comparison could be achieved using a brute force pixel-wise comparison or by extracting features from images and comparing them. Depending on the requirements such as sensitivity to scale and rotation, size of images, etc. one method or the other could be preferable. In the case of single particle imaging the detector is placed at a fixed distance at a fixed orientation, therefore, either approach is applicable. The advantage of comparing features over brute force approach is the reduced number of comparisons. Comparing N datasets to M reference models, each with $n\ x\ n$ pixels is an operation with quadratic complexity, $O(N^2)$. Using $f\ (f < n\ x\ n)$ features reduces the computations involved in each comparison. We tried a number of feature extraction approaches that will be described in the Technical Approach section.

Orientation matching involves reading many images and references from files. The image data and references are stored in HDF5 format and reading this data from the files added up to 25 – 120% of the computation time depending on the data sizes. Therefore, along with improving the computational performance of orientation matching, the HDF5 reads require optimization.

2. MILESTONE OVERVIEW

This milestone primarily involved accelerating orientation matching for use in the M-TIP SPI algorithm which reconstructs the 3D structure of molecules. We implemented a number of different algorithms for orientation matching both on CPU (for baseline and comparison) and GPU (for acceleration) and also optimized the image data reads. Completion criteria for this milestone is GPU-accelerated orientation matching kernel code running on Summit and Perlmutter (AKA NERSC-9, but now represented by Cori GPU nodes), this report and a highlight slide. Predecessor dependency is the FY20 milestone "ADSE13-198 Realistic simulation of SPI data accounting for beam features, sample injector and detector, for M-TIP to ingest." Successor dependencies are FY20 milestone "ADSE13-193 Scaling single particle imaging with M-TIP on Summit (simulated data)" and FY21 milestones "ADSE13-188 Single particle imaging with M-TIP at scale on Perlmutter and Summit (noisy simulated data)" and "ADSE13-179 Run M-TIP against realistic simulated data."

2.1 DESCRIPTION

This milestone primarily involved the orientation matching step of the M-TIP SPI algorithm. Our task was to port this step to GPUs, benchmark, optimize and down select. For this acceleration work we used both Summit and Cori GPUs.

2.2 EXECUTION PLAN

- Obtain Python orientation matching code and synthetic data
- Evaluate and benchmark initial code
- Plan acceleration effort (language, modular design, etc.)
- Execute port to GPU
- Benchmark and optimize.
- Port to Summit and Perlmutter when available.

2.3 COMPLETION CRITERIA

Completion criteria is the code [2], this report and a highlight slide.

2.4 MILESTONE DEPENDENCIES

2.4.1 Milestone Predecessors

Predecessor dependency is the FY20 milestone "ADSE13-198 Realistic simulation of SPI data accounting for beam features, sample injector and detector, for M-TIP to ingest."

2.4.2 Milestone Successors

Successor dependencies are FY20 milestone "ADSE13-193 Scaling single particle imaging with M-TIP on Summit (simulated data)" and FY21 milestones "ADSE13-188 Single particle imaging with M-TIP at scale on Perlmutter and Summit (noisy simulated data)" and "ADSE13-179 Run M-TIP against realistic simulated data."

3. TECHNICAL WORK SCOPE, APPROACH, RESULTS

3.1 Work Scope

The scope of this milestone is to develop the GPU acceleration of the orientation matching. The I/O performance for reading the images is also included, because it was found to have an impact.

3.2 Datasets

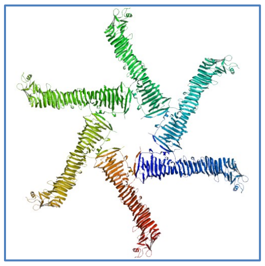
Some background on image data for SPI.

- Size: Typically, the detector pixels are binned by a factor of 4 (or 8). One can expect an image size of 256x256 (or 128x128). Future LCLS2 detectors may be larger (512x512).
- Number of data images: Big particles like viruses at low resolution scatter many photons, hence require less snapshots, so ~3k images are typical. LCLS2 will look at small proteins at high resolution, so one can expect around 600k images. Refer to Table 1 of [3]. Numbers of images can be multiplied by 10x to 50x if studying conformations and heterogeneity.
- Number of reference images: Depending on the noise, you could assume 2x~10x less than number of data images.

For our approach, we used two datasets, one smaller and one larger.

These datasets include "data" images (XRD at unknown orientations) and "reference" images, which are images of XRD for which we know the orientation. Datasets are available from ExaFEL SLAC staff upon request.

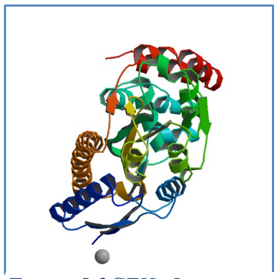
6NYF-Helicobacter pylori Vacuolating Cytotoxin A Oligomeric Assembly 1 (OA-1) [4] (simulated via a cryoEM simulation)



This data set has 19996 data images and 3599 reference images, represented as single precision - float32. The 6NYF data used is from a cryoEM simulation that was just noisy enough so that the particles were not visible to the eye. All the comparisons across libraries and the LANL CUDA codes are made using this dataset as it fits in a single GPU.

Figure 4 6NYF. Image courtesy [4].

2CEX-Structure of a sialic acid binding protein (SiaP) in the presence of the sialic acid analogue Neu4Ac2en [5] (simulated by ExaFEL's XRD simulator, pysingfel)



This data set had a total of 500,000 images. They are partitioned into 400,000 data images and 100,000 reference images. The 2CEX data used is simulated noise-free data created using the **pysingfel** simulation code which was created for the ExaFEL project in milestone ADSE13-198. The multi-GPU multi-node implementation scaled to the larger number of nodes on Summit uses this dataset.

Figure 5 2CEX. Image courtesy [5].

3.3 Euclidean Distance and K-Nearest Neighbors

The Euclidean distance is computed across each data image with respect to each reference image. The Euclidean distance between a data images and a reference image with n-dimensions can be represented as

$$dist = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

Where, $a = (a_1, a_1, \dots, a_n)$ is a data image, and $b = (b_1, b_1, \dots, b_n)$ is a reference image.

For a dataset with m data image and n reference images with d dimensions, the matrix dimensions of the data images is $m \times d$, reference images is $n \times d$, and the computed Euclidean distance matrix is $m \times n$. So, the value at row x and column y in the Euclidean distance matrix is the computed Euclidean distance of the data image present at row x in the data images matrix, and of the reference image present at the row y in the reference images matrix.

The weighted Euclidean distance incorporates the feature weights w on each dimension.

$$dist = \sqrt{\sum_{i=1}^{n} w_i(a_i - b_i)^2}$$

For noiseless data some kind of weighted Euclidean distance is usually good. With noise, a distance-based on maximum likelihood is typically optimal, and can often be represented by weighting the Euclidean distance by the variance of the noise.

The k-Nearest Neighbor (kNN) algorithm is used to compute the k-nearest neighbors for each data image across the reference images. The kNN algorithm involves computing Euclidean distance for various data images with respect to the reference images, and uses a sorting algorithm (such as heapsort) to sort the distances.

3.4 Sequential and multi-threaded Orientation Matching Algorithms and Implementations

Sequential Python v1

This version uses single-threaded Scikit-learn's [6] Euclidean distance library. The data and the reference images are read into memory once. Ten nearest neighbors are computed.

Sequential Python v2

This version uses single-threaded Scikit-learn's KNeighborsClassifier library. The ten nearest neighbors are computed using brute-force search option to compute nearest neighbors.

Scale-Invariant Feature Transform (SIFT)

Scale-Invariant Feature Transform (SIFT) [7] is used in computer vision to detect and describe local features in images. SIFT extracts large collection of feature vectors that are invariant to image translation, scaling, and rotation, partially invariant to illumination changes and robust to local geometric distortion using the following major computation stages: scale-space extrema detection, keypoint localization, orientation assignment, and keypoint descriptor. SIFT keypoints (features) can be extracted for both data and reference images, which can be individually compared to find similarity. We used the fast variant of SIFT known as Speeded-UP Robust Features (SURF) [8], which approximates some of the computations in SIFT for faster feature extraction. However, the feature extraction time for 1500 data images and 600 reference images with 384 x 384 pixels each was around 148 s. This was much higher than the brute force method of pixel-wise comparison, which took around 6 s for the same data set and hence was not pursued. In addition, the scale and rotational invariance offered by SIFT is not necessary as the detector is placed at a fixed distance at a fixed orientation throughout the experiment.

Feature extraction using Deep-Neural Networks

In order to reduce the time taken for feature extraction, we tried using neural networks. One of the problems associated with deep neural networks is that they are more difficult to train. Therefore, techniques like residual learning [9] are used to ease the training of networks that are substantially deeper than those used previously. However, for the purpose of our problem, we just need to extract the features in images rather than classifying them into categories. Hence, we used transfer learning, where a pre-trained Resnet-50 model trained on the ImageNet dataset was adopted. This dataset takes images of size 224 x 224 and extracts 1000 features for each image. Even this approach took a substantial 111 s for feature extraction for 1500 images and 600 references and hence was discontinued.

Histogram feature matching

As we saw in the previous two approaches, the feature extraction time was significantly higher compared to the brute-force pixel-wise comparison. So, we started looking for a very simple feature extraction strategy and tried the histogram-based approach. In this approach, we computed the histograms of the data and reference images organized into 256 bins and computed the correlation between the histograms as a measure of similarity. Even though this approach was faster than the previous two approaches, it was still slightly slower than the brute-force approach.

FLANN C OpenMP

We tried a number of OpenMP orientation matching codes using the C-based FLANN scalable nearest neighbors library [10,11]. Of the nearest neighbor search algorithms, we tried Linear (brute force), KDtree, Autotuned and Kmeans. We found Linear to be the fastest and Kmeans was not used because the outputs were different from what was expected (compared to brute force).

FAISS C OpenMP

This version using OpenMP was significantly faster than the FLANN versions.

3.5 CUDA-Based GPU AcceleratеD Explorations of Orientation Matching Algorithms and Implementations

The proposed orientation matching implementation is a k-Nearest Neighbor (kNN) algorithm, which is used to compute the k-nearest neighbors for each data image across the reference images. The kNN algorithm involves computing Euclidean distance for various data images with respect to the reference images, and a heap sort algorithm implementation. In the proposed method, both the Euclidean distance and the heap sort implementations are developed using CUDA programming model.

3.5.1 CrabCUDA

CrabCUDA [12] is an open source Python package that implements a GPU accelerated version of Scikitlearn. As the idea of feature extraction was proven to be computationally expensive from our testing, we tried using the accelerated version of the brute-force comparisons. The GPU-accelerated Python version using CrabCUDA is about twice as fast as the Scikitlearn CPU version, and is slower than other GPU implementations discussed in later sections.

3.5.2 cuML

CuML [13] is developed for the NVIDIA Rapids library. This Python version of orientation matching, uses the GPU version of “k-neighbors” and is much slower than the FAISS and LANL CUDA version on a single node.

3.5.3 FAISS CUDA Version

FAISS [14] is Facebook’s library for similarity search for very large datasets. The FAISS library uses variant of batcher's bitonic sorting [15] and IVFADC indexing [16].

3.5.4 Background to LANL CUDA Versions

CUDA Tiling Method

In the CUDA tiling method, tiles of data from both the data images and the reference images are copied into shared memory one at a time. A local Euclidean distance is computed for each tile and the distances are summed. Each element in the Euclidean distance is computed in each thread of the CUDA core. The tiling of the data can be represented as shown in Figure 6. m is the number of data images, n is the number of reference images and d is the size of each image.

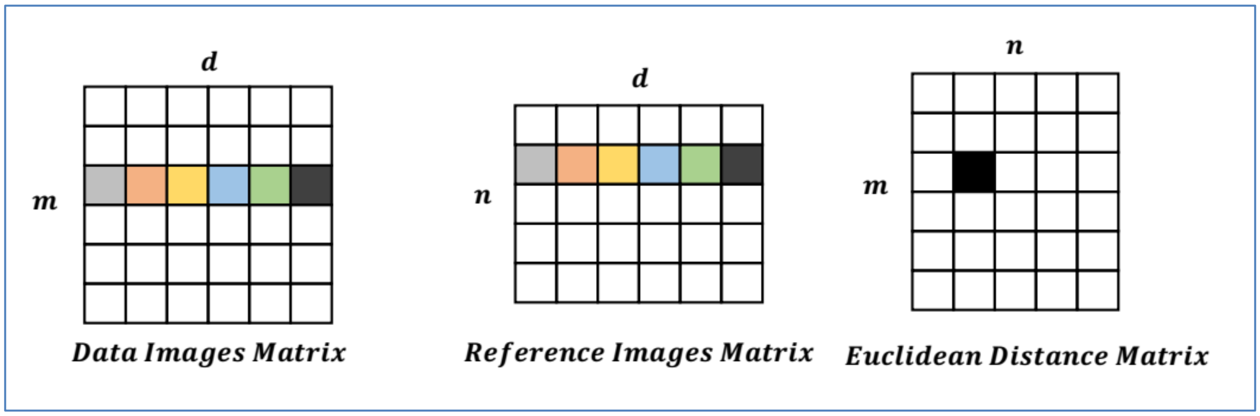


Figure 6 Illustration of the CUDA tiling method.

The basic CUDA implementation kernel is summarized in the algorithm below:

```
$\textbf{Input}$: $data\ images\ matrix (m \times d), reference\ images\ matrix\ (n$
                $\times\ d), CUDA\ blocks\ with\ TILE\ number\ of\ threads\ in\ x - direction.$
$\textbf{Output}$: $Euclidean\ Distance\ (m \times n)$
$\textbf{Implementation}$:
$Initialize\ CUDA\ thread\ and\ block\ indices\ txId, tyId, bxId, byId\ in\ x\ and\ y - direction.$
$Allocate\ 2D\ shared\ memories\ with\ size\ of\ number\ of\ CUDA\ threads\ per\ block$
$Initialize\ variables\ tmpResult = 0, and\ diff = 0$
$\textbf{Do}\ iterationId = 0, untill\ iterationId < d/TILE$
    $load\ iterationId\ tile\ from\ data\ matrix\ into\ shared\ memory\ of\ sharedData$
    $load\ iterationId\ tile\ from\ reference\ matrix\ into\ shared\ memory\ of\ sharedReference$
    $synchronize\ threads$
    $\textbf{Do}\ tileItr = 0, untill\ tileItr < TILE$
        $diff = (sharedData[tyId][tileItr] - sharedReference[txId][tileItr])$
        $tmpResult += diff * diff$
    $\textbf{End do}$
    $synchronize\ threads$
    $store\ the\ tmpResult\ in\ the\ euclidean\ distance\ matrix$
$\textbf{End do}$
```

CUDA Heap Sort Implementation

The Euclidean distances computed need to be sorted to identify the minimum distance reference images. Many sorting algorithms exist for finding out the minimum values in the data. In this implementation, the heap sort algorithm from [14] is ported onto GPU using the CUDA programming model. Each thread in the CUDA blocks computes the heap sort algorithm for one data image. As a result, a total of data images number of threads are invoked in CUDA.

On a single GPU implementation, since the Euclidean distances with respect to all the reference images are computed on the same GPU, no data communications are required. But in a multi-GPU implementation, a different set of reference images are used in the GPUs in the x-direction with respect to each data image. As a result, data communication is required across GPUs which compute the Euclidean distance for the same data images (across GPUS in the same row of the multiple-GPU layout). But transferring the entire Euclidean distance matrix computed in each GPU will result in data transfers with a huge amount of data. Instead, each GPU computes the Euclidean distance and the heap sort algorithm on the local data images and the reference images. Then only the local k-Nearest Neighbors of the data images is transferred to the first GPU in the row, and the first GPU once again computes the k-Nearest Neighbors across the data obtained from other GPUs.

The data communication between GPUs can be represented as below in Figure 7. The GPU1, GPU 2, GPU 3 which are in the same row will have the same data images but different reference images. Similarly, GPU 1, GPU 4 and GPU 7 which are in the same column will have the same reference images but different data images. The local k-Nearest Neighbors of the data images are transferred to the first GPU in the row, and the first GPU once again computes the k-Nearest Neighbors across the data obtained from other GPUs.

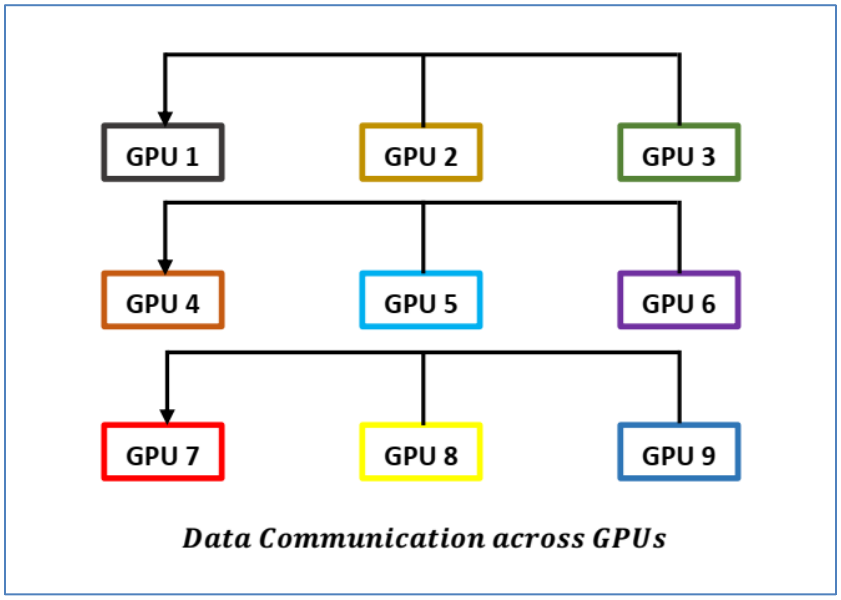


Figure 7 Data communication between GPUs for k-Nearest Neighbots algorithm

3.5.5 LANL CUDA v3 and v5 Implementations

Previous explorations using available CUDA libraries showed that this direction is promising so we focused our efforts on implementing our own faster version of the orientation matching in CUDA. After implementing a few CUDA versions which we call LANL CUDA vX, the most promising were versions v3 and v5. We will not detail v1, 2 and 4 in this report.

The LANL CUDA v3 and v5 implementations are an extension to the basic CUDA kernel implementation, also using the CUDA tiling method and heapsort. In the basic CUDA implementation, each CUDA thread computes the Euclidean distance between one data image and one reference image. This is done by iteratively copying data into shared memory and computing the $L2$ norm (distance) locally. In order to compute image similarly, only the squared difference is sufficient. Hence, we omit the square root to reduce the total number of computations performed. In the CUDA memory hierarchy, the memory access latency is as follows: global memory > shared memory > registers. Therefore, the data load transactions are reduced as the data is copied into shared memory using a coalesced memory access pattern for optimal memory accesses, and shared memory is used in the computations.

In the LANL CUDA v3 and v5 implementations, we take advantage of the register memory by copying data from shared memory to register memory. In addition, unrolling is used to compute multiple

Euclidean distance calculations per thread, which increases the number of in-flight instructions, thereby, resulting in betting instruction level parallelism. This also results in more independent memory load/store operations and effective register utilization in a single thread to yield better performance as memory latency can be hidden. The reference images are arranged in the x-direction of the grid, and the data images in the y-direction. Each thread in the logical x-direction of the thread block computes the Euclidean distance for reference images with respect to one data image and vice-versa.

In LANL CUDA v3, we compute the Euclidean distance between 4 data images and 4 reference images per thread, resulting in a 4x4 distance matrix. In contrast, the LANL CUDA v5 uses 6 data and 6 reference images. In addition, the LANL CUDA v5 is a multi-GPU implementation targeted towards larger datasets, where the Euclidean distance for a group of images is computed per GPU. The data distribution scheme of reference images along the logical x-direction and data images along the logical y-direction is extended across multiple GPUs as shown in Figure 8. m is the number of data images and n is the number of reference images

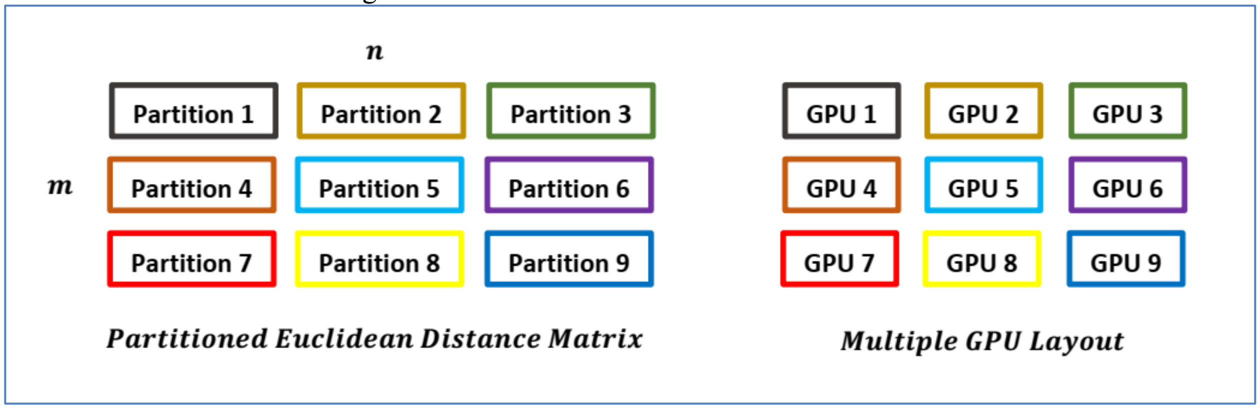


Figure 8 Data distribution scheme for reference images.

In this data layout, each GPU receives a third of data and reference images and Euclidean distances can be computed independently without requiring any communication between GPUs.
We implemented two versions of the LANL CUDA vX codes to compute the weighted and unweighted Euclidean distances. LANL CUDA v3 performs better than v5 for weighted Euclidean distances, whereas, v5 is better for unweighted Euclidean distances. The results shown in this report using different available libraries and single GPU cuda implementation uses unweighted Euclidean distance and only the multi-GPU LANL CUDA v3 uses weighted Euclidean distance. If weighted Euclidean distance is preferred, one could pre-weight the data and reference images with the unweighted versions.

3.6 Summary of Orientation Matching Algorithms Explored and Implemented

Table 1 Summary of algorithms

Algorithm or Version	Execution Type	Origin	Notes
Brute Force	Sequential	Python sklearn - euclidean_distances and numpy - argpartition	Pixel-wise comparison; unweighted
Brute Force	Sequential	Python sklearn - k-nearest neighbors (KNN)	Pixel-wise comparison with KNN matching; unweighted
Scale-Invariant Feature Transform (SIFT)	Sequential	OpenCV library	computer vision approach to extract features; unweighted
Deep-Neural Networks (DNN)	Sequential	ResNet-50 model	Feature extraction using transfer learning
Histogram feature matching	Sequential	original	Compute correlation between histograms
FLANN C OpenMP Linear, KDTree, Autotuned and Kmeans	Multi-threaded CPU	FLANN library	Fastest FLANN was Linear. K-means didn't match base output; unweighted
FAISS OpenMP	Multi-threaded CPU	FAISS library	Fastest OpenMP version tested
CrabCUDA	CUDA GPU	CrabCUDA library	CUDA version of Skikit-learn, 2x faster than sequential; unweighted
CuML	CUDA GPU	NearestNeighbors library from CuML	unweighted
FAISS GPU	CUDA GPU	FAISS library	Next fastest to LANL CUDA; unweighted
LANL CUDA v3 (weighted and unweighted versions)	CUDA GPU	original	Euclidean distance between 4 data images and 4 reference images per CUDA thread. Better for weighted.
LANL CUDA v5 (weighted and unweighted versions)	CUDA GPU	original	Euclidean distance between 6 data images and 4 reference images

			per CUDA thread. Better for unweighted.

3.7 I/O Optimization

The image data and references are stored in HDF5 format and reading this data from the files added up to 25 – 120% of the computation time depending on the data sizes. Therefore, along with improving the computational performance of comparisons we also optimized the HDF5 reads.

One of the first approaches we tried was to use multiple MPI processes to access the HDF5 file, which improved the data read time by a factor of 1.5 – 5 depending on the file size. Later, the files were broken into smaller parts and accessed using multiple MPI processes.

3.8 Results

3.8.1 Orientation Matching

Library Comparison on Cori

Below in Figure 9 we compare the various libraries we tried: sequential (on single core), multi-threaded FAISS (40 threads and fastest multi-threaded version that we tested), GPU FAISS (on one GPU) and LANL CUDA v3 (on one GPU). All tests were done on Cori GPU nodes and used unweighted Euclidean distance. Tests used the 6NYF dataset.

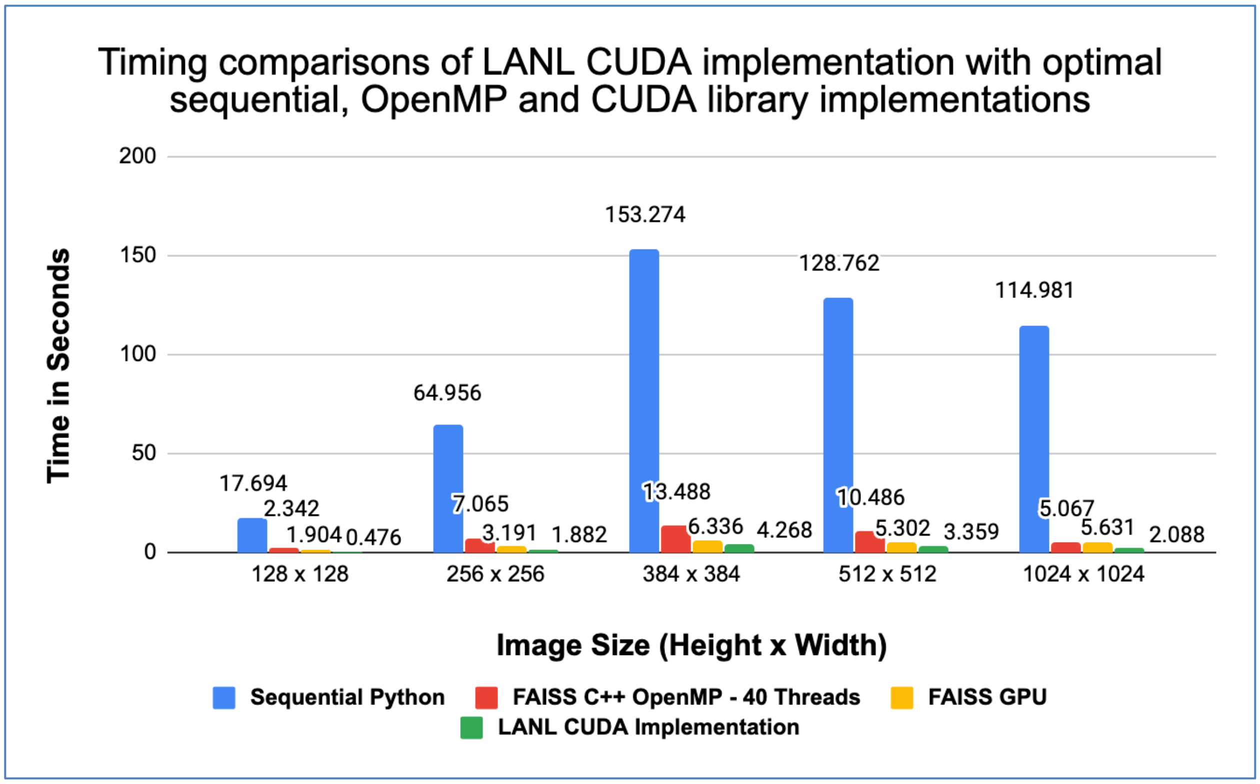


Figure 9 Comparison of various implementations of orientation matching algorithm on different image sizes.

Notes: All the 128x128, 256x256 and 384x384 size images are timed with 19996 data images and 3599 reference images. The 512x512 size images are timed with 11000 data images and 2500 reference images due to a memory limit on GPUs. The 1024x1024 size images are timed with 2500 data images and 1000 reference images due to a memory limit on GPUs.

Conclusions: GPU (FAISS and LANL CUDA) implementations offer speedup over sequential and multi-threaded implementations. **The LANL CUDA v3 implementation is 34-55X faster than sequential, 2.4-4.9X faster than FAISS OpenMP and 1.5-4X faster than FAISS GPU**.

Cori Versus Summit Single GPU LANL CUDA Version

Below in Figure 10 and 11 we measure LANL CUDA v5 (unweighted) and v3 (weighted) implementation on a single GPU on both Cori GPU and Summit. We use the 6NYF dataset. Notes: All the 128x128, 256x256 and 384x384 are timed with 19996 data images and 3599 reference images. The 512x512 is timed with 11000 data images and 2500 reference images due to the memory limit on GPUs. The 1024x1024 is timed with 2500 data images and 1000 reference images due to the memory limit on GPUs.

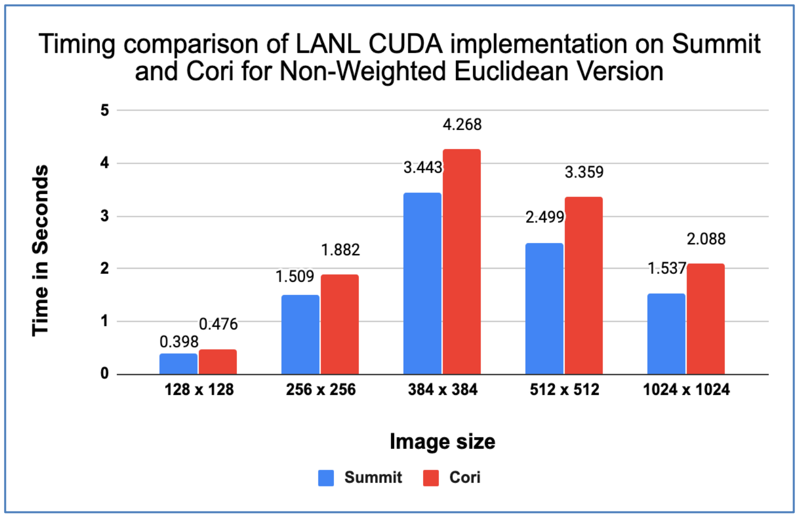


Figure 10 Performance of LANL CUDA unweighted algorithm on Summit and Cori

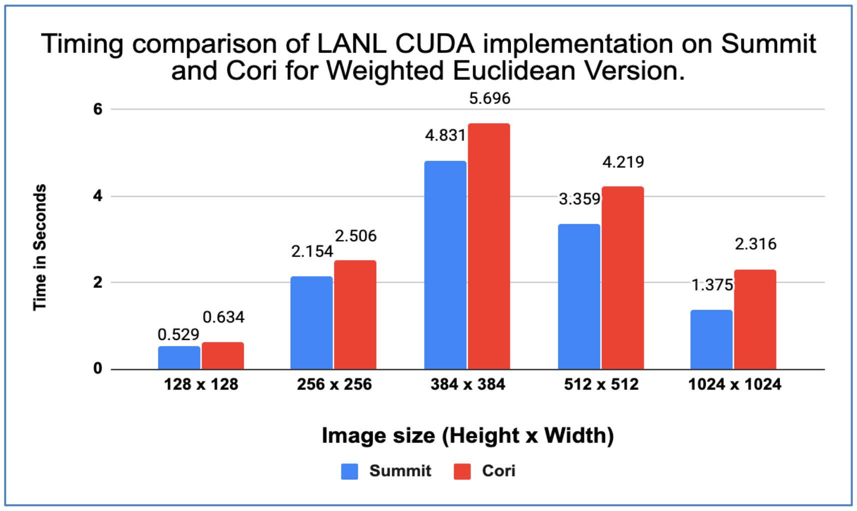


Figure 11 Performance comparison of LANL CUDA weighted algorithm on Summit and Cori

Conclusion: We measured faster overall performance on Summit. NVlink on Summit reduces the CPU-GPU communication compared to the PCIe Gen3 on Cori GPU nodes. Not shown here, but using pinned memory on the host side reduces the CPU-GPU data transfer time by a factor of 3. Also, image size plays a role; mid-range image sizes seem to take more time.

Multi-node, Multi-GPU LANL CUDA Version Strong Scaling on Summit

Figure 12 shows strong scaling of LANL CUDA implementation on Summit, with partitioned data and reference images. The dataset used here was the larger dataset, 2CEX, produced by the pysingfel simulation.

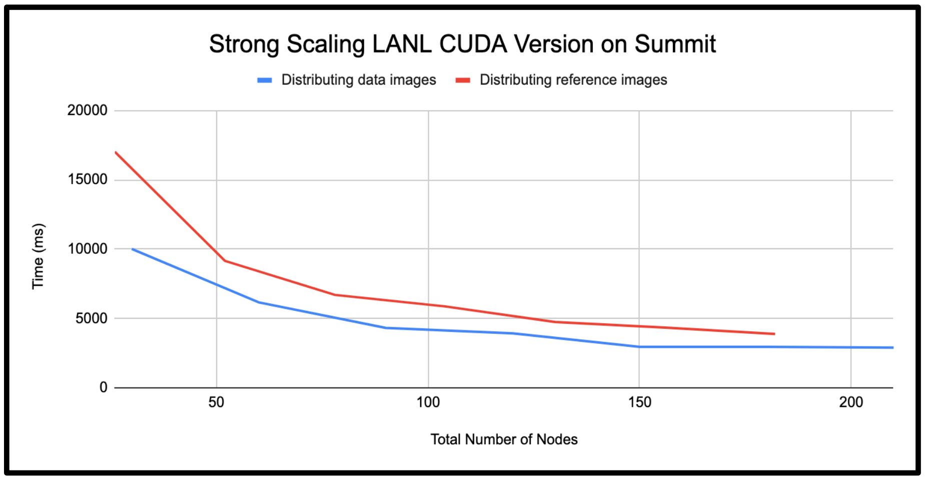


Figure 12 Strong scaling of LANL CUDA version on Summit.

Conclusion: The LANL CUDA multi-node, multi-GPU implementation shows mostly linear strong scaling. Although multi-node scaling was not required for this milestone, we felt it should be measured to show our multi-GPU multi-node implementation of orientation matching.

MPI communication is similar with data images. When you partition the reference images, the MPI communication increases. Communication is explained in Section 3.5.3. Reference images are spread across columns so that increases communication time. GPUs need to communicate with rank 0 of that row.

Figure 13 shows the LANL CUDA multi-rank single-node weighted Euclidean distance version with the 6NYF dataset on Summit.

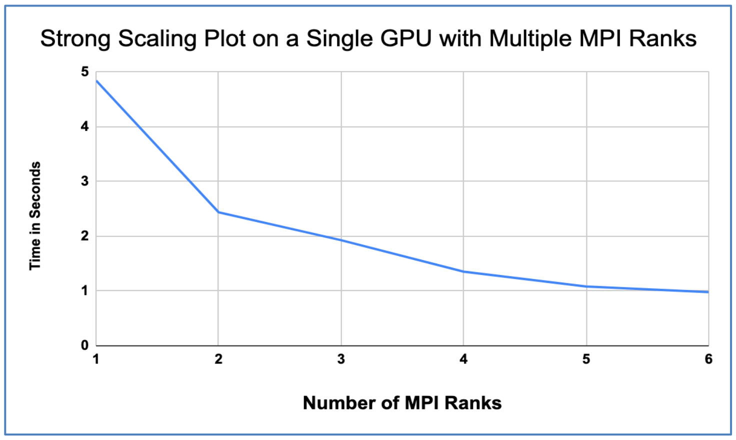


Figure 133 Strong Scaling of LANL CUDA version over MPI Ranks on a single node.

3.8.2 HDF5

Figure 14 shows the graph of the read time for a single HDF5 file compared to the data split into 4 files for varying number of MPI processes. This test was performed on a single Haswell node of Cori.

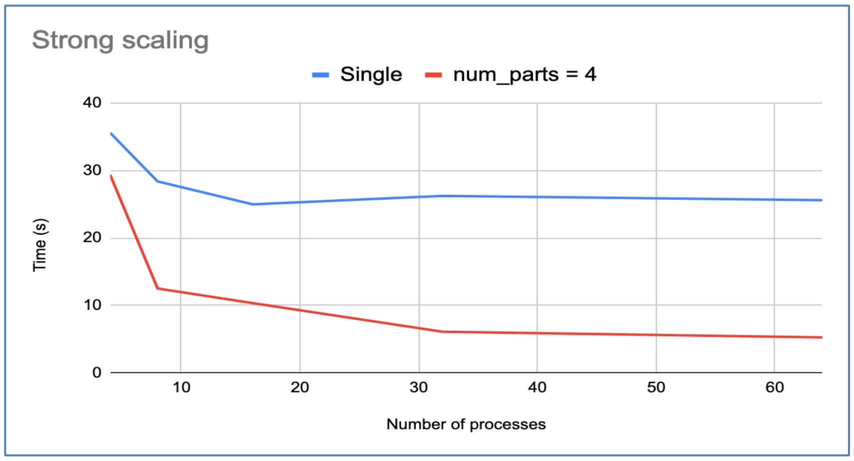


Figure 144 HDF5 read times with increasing MPI processes.

The effect of the number of file parts was also studied and the results are shown below in Figure 15. For this run 64 MPI processes were used in total and the number of file parts were varied. We can see that the read time improves with increasing number of file parts.

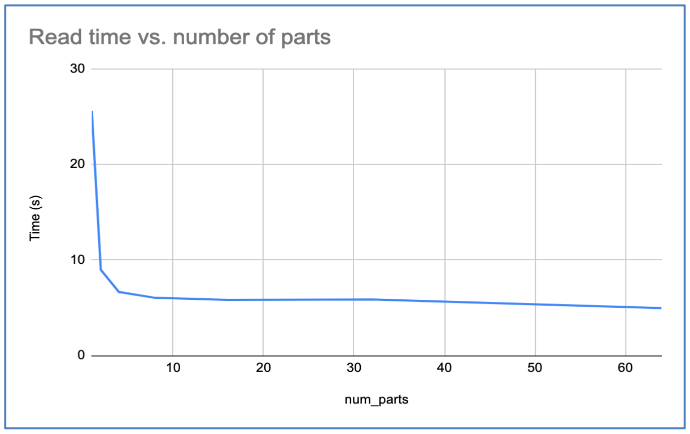


Figure 15 HDF5 read time with increasing number of file parts.

In order to improve the I/O performance, we made use of burst buffers available on Summit and Cori. Burst buffers (BB) are technologies that provide faster I/O based on new solid-state media. On Summit, each compute node has a Samsung PM1725a NVMe SSD of capacity 1.6 TB. Cori has 1.8 PB of shared burst buffer based on Cray DataWarp that uses flash or SSD technology. The results are summarized in Fig. 16 and Fig. 17. We can see that on Summit, with individual BB per compute node, there is ~85x improvement in HDF5 read times.

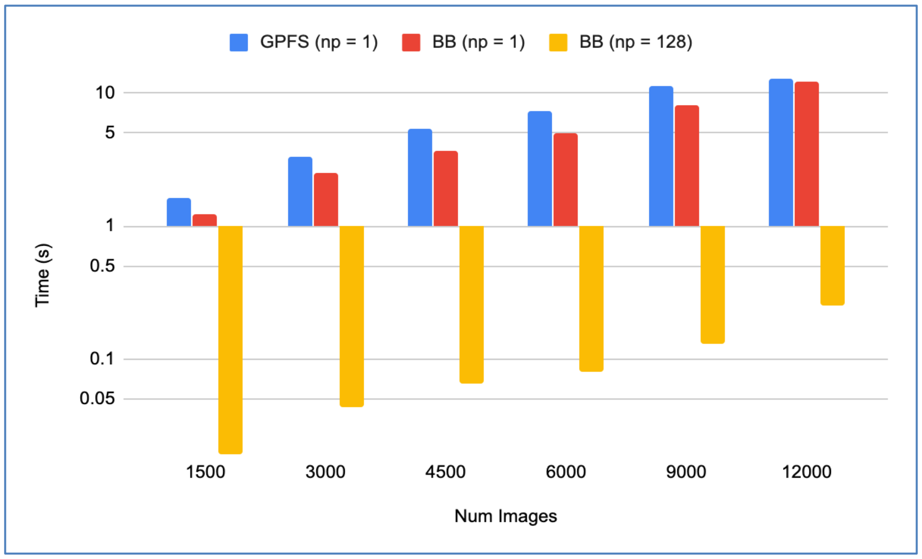


Figure 16 HDF5 read times on Summit.

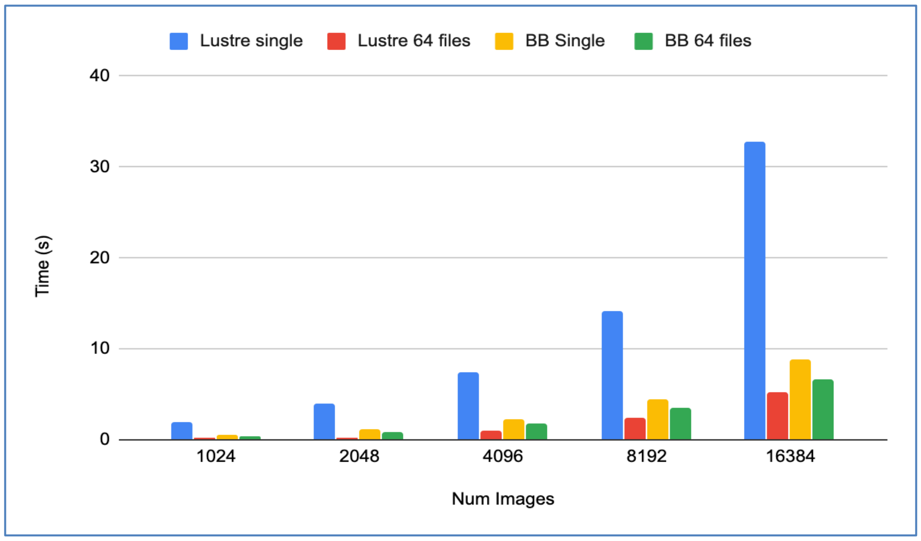


Figure 157 HDF5 read times on Cori (np=64)

4. RESOURCE REQUIREMENTS

Person-months for this milestone were 13 person-months. Perlmutter (Cori GPU) hours were 100, Cori hours were 10,549 and Summit hours were 3390.

5. CONCLUSIONS AND FUTURE WORK

We have accelerated the orientation matching portion of the single particle imaging M-TIP image reconstruction algorithm on GPU on Cori and Summit. The LANL GPU v3 implementation is 34-55X faster than sequential, 2.4-4.9X faster than FAISS OpenMP and 1.5-4X faster than FAISS GPU. Summit single-node GPU versions were somewhat faster than Cori GPU. Image size plays a role; mid-range image sizes take more time. The LANL CUDA multi-node, multi-GPU implementation shows linear strong scaling. I/O also plays a large role; splitting data into parts improves read time and burst buffers dramatically improve read times.

In the near term we will be looking at performance portability for this code on upcoming exascale platforms (Aurora and Frontier) via using their test clusters (Iris and Tulip). We will also incorporate this scalable multi-GPU and multi-node orientation matching implementation into the cartesian-based M-TIP algorithm. In the process of completing this milestone, we also investigated the impact of HDF5 read times for the input data to orientation matching and concluded that burst buffers will significantly improve performance of the algorithm.

6. ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations—the Office of Science and the National Nuclear Security Administration—responsible for the planning and preparation of a capable exascale ecosystem—including software, applications, hardware, advanced system engineering, and early testbed platforms—to support the nation's exascale computing imperative.

We would like to acknowledge the following team members for their contributions:

Pranay Kommera, LANL – GPU acceleration and benchmarking
Vinay Ramakrishnaiah, LANL - GPU acceleration and benchmarking, I/O optimization
Jeffrey Donatelli, LBL – consulting on algorithm, datasets
Petrus Zwart, LBL – consulting on algorithm
Chuck Yoon, SLAC – initial orientation matching code and synthetic data, consulting regarding integration with other components
Antoine Dujardin, SLAC – consulting regarding future integration with other M-TIP components
Christine Sweeney, LANL - lead

7. REFERENCES

1. J. J. Donatelli, P. H. Zwart and J. A. Sethian. Iterative phasing for fluctuation X-ray scattering. Proceedings of the National Academy of Science. USA 112, 10286-10291, 2015.
2. P. Kommera. LANL CUDA versions of orientation matching code. https://gitlab.osti.gov/pkommera/orientation-matching
3. Poudyal, I et al. "Single-particle imaging by x-ray free-electron lasers-How many snapshots are needed?." *Structural dynamics (Melville, N.Y.)* vol. 7,2 024102. 20 Mar. 2020, doi:10.1063/1.5144516
4. 6NYF-Helicobacter pylori Vacuolating Cytotoxin A Oligomeric Assembly 1 (OA-1)**.** https://www.rcsb.org/structure/6NYF
5. 2CEX-Structure of a sialic acid binding protein (SiaP) in the presence of the sialic acid analogue Neu4Ac2en. https://www.rcsb.org/structure/2CEX
6. Raschka, Sebastian and Patterson, Joshua and Nolet, Corey. "Machine Learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence." arXiv:2002.04803, 2020.
7. Lowe, David G. "Distinctive image features from scale-invariant keypoints." *International journal of computer vision* 60.2 (2004): 91-110.
8. Bay, Herbert, et al. "Speeded-up robust features (SURF)." *Computer vision and image understanding* 110.3 (2008): 346-359.
9. He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.
10. Muja, Marius, and David G. Lowe. "Scalable nearest neighbor algorithms for high dimensional data." *IEEE transactions on pattern analysis and machine intelligence* 36, no. 11 (2014): 2227-2240.
11. Muja, Marius, and David Lowe. "Flann-fast library for approximate nearest neighbors user manual." *Computer Science Department, University of British Columbia, Vancouver, BC, Canada* (2009).
12. CrabCUDA. https://github.com/vinigracindo/crabcuda
13. CuML. https://github.com/rapidsai/cuml
14. (FAISS) Johnson, Jeff, Matthijs Douze, and Hervé Jégou. "Billion-scale similarity search with GPUs." *IEEE Transactions on Big Data* (2019).
15. K. E. Batcher. Sorting networks and their applications. Spring Joint Computer Conference, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

16. H. J´egou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. IEEE Trans. PAMI, 33(1):117–128, January 2011.
17. Heapsort. https://www.geeksforgeeks.org/heap-sort/